Unstructured Overlapping Mesh Distribution in Parallel

MATTHEW G. KNEPLEY, Rice University

MICHAEL LANGE, Imperial College London

GERARD GORMAN, Imperial College London

We present a simple mathematical framework and API for parallel mesh and data distribution, load balancing, and overlap generation. It relies on viewing the mesh as a Hasse diagram, abstracting away information such as cell shape, dimension, and coordinates. The high level of abstraction makes our interface both concise and powerful, as the same algorithm applies to any representable mesh, such as hybrid meshes, meshes embedded in higher dimension, and overlapped meshes in parallel. We present evidence, both theoretical and experimental, that the algorithms are scalable and efficient. A working implementation can be found in the latest release of the PETSc libraries.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: *Parallel and vector implementations*; G.1.8 [**Numerical Analysis**]: Partial Differential Equations—*Finite Element Methods*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: mesh distribution, mesh overlap, Hasse diagram, CW complex, PETSc

ACM Reference Format:
Matthew G. Knepley, Michael Lange, and Gerard J. Gorman, 2014. Unstructured Overlapping Mesh Distribution in Parallel. ACM Trans. Math. Softw. 0, 0, Article 0 (2014), 14 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The algorithms and implementation for scalable mesh management, encompassing partitioning, distribution, rebalancing, and overlap generation, as well as data management over a mesh can be quite complex. It is common to divide meshes into collections of entities (cell, face, edge, vertex) of different dimensions which can take a wide variety of forms (triangle, pentagon, tetrahedron, pyramid, ...), and have query

MGK acknowledges partial support from DOE Contract DE-AC02-06CH11357 and NSF Grant OCI-1147680. ML and GJG acknowledge support from EPSRC grant EP/L000407/1 and the embedded CSE programme of the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk). All authors acknowledge support from the Intel Parallel Computing Center program through grants to both the University of Chicago and Imperial College London.
Authors' addresses: M.G. Knepley, Computational and Applied Mathematics, Rice University, Houston, TX; email: knepley@rice.edu; M. Lange, Imperial College London; email: michael.lange@imperial.ac.uk; G.J. Gorman, Imperial College London; email: g.gorman@imperial.ac.uk

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

functions tailored to each specific form [D'Azevedo and et. al. 2015]. This code structure, however, results in many different cases, little reuse, and greatly increases the complexity and maintenance burden. On the other hand, codes for adaptive redistribution of meshes based on parallel partitioning such as the Zoltan library [Devine et al. 2006], usually represent the mesh purely as an undirected graph, encoding cells and vertices and ignoring the topology. For data distribution, interfaces have been specialized to each specific function space represented on the mesh. In Zoltan, for example, the user is responsible for supplying functions to pack and unpack data from communication buffers. This process can be automated however, as in DUNE-FEM [Dedner et al. 2010] which attaches data to entities, much like our mesh points described below.

We have previously presented a mesh representation which has a single entity type, called *points*, and a single antisymmetric relation, called *covering* [Knepley and Karpeev 2009]. This structure, more precisely a Hasse diagram [Birkhoff 1967; Wikipedia 2015b], can represent any CW-complex [Hatcher 2002; Wikipedia 2015a], and can be represented algorithmically as a directed acyclic graph (DAG) over the points. It comes with two simple relational operations, $\mathtt{cone}(p)$, called the *cone* of p or the in-edges of point p in the DAG, and its dual operation $\mathtt{supp}(p)$, called the *support* of p or the out-edges of point p. In addition, we will add the transitive closure in the DAG of these two operations, respectively the closure $\mathtt{cl}(p)$ and star $\mathtt{st}(p)$ of point p. In Fig. 1, we show an example mesh and its corresponding DAG, for which we have $\mathtt{cone}(A) = \{a, b, e\}$ and $\mathtt{supp}(\beta) = \{a, c, e\}$, and the transitive closures $\mathtt{cl}(A) = \{A, a, b, e, \alpha, \beta, \gamma\}$ and $\mathtt{st}(\beta) = \{\beta, a, c, e, A, B\}$.

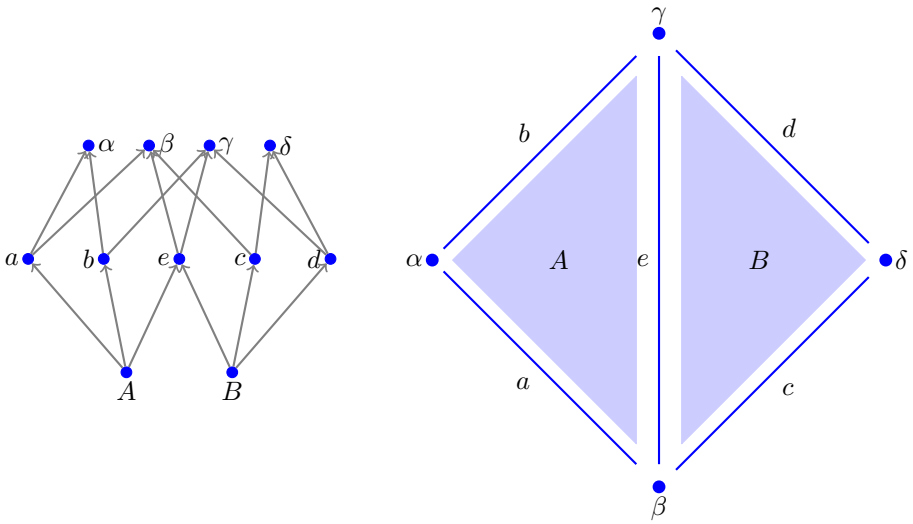

Fig. 1. A simplicial doublet mesh and its DAG (Hasse diagram).

In our prior work [Knepley and Karpeev 2009], it was unclear whether simple generic algorithms for *parallel* mesh management tasks could be formulated, or various types of meshes would require special purpose code despite the generic mesh representation. Below, we present a complete set of generic algorithms, operating on our generic DAG representation, for parallel mesh operations, including partitioning, distribution, re-

balancing, and overlap generation. The theoretical underpinnings and algorithms are laid out in Section 2, and experimental results detailed in Section 3.

2. THEORY

2.1. Overlap Creation

We will use the Hasse diagram representation of our computational mesh [Knepley and Karpeev 2009], the `DMPlex` class in PETSc [Balay et al. 2014a; Balay et al. 2014b], and describe mesh relations (adjacencies) with basic graph operations on a DAG. A distributed mesh is a collection of closed serial meshes, meaning that they contain the closure of each point, together with an "overlap structure", which marks a subset of the mesh points and indicates processes with which these points are shared. The default PETSc representation of the overlap information uses the `SF` class, short for *Star Forest* [Brown 2011]. Each process stores the true owner (root) of its own ghost points (leaves), one side of the relation above, and construct the other side automatically.

In order to reason about potential parallel mesh algorithms, we will characterize the contents of the overlap using the mesh operations. These operations will be understood to operate on the entire parallel mesh, identifying shared points, rather than just the local meshes on each process. To indicate a purely local operation, we will use a subscript, e.g. $\mathbb{cl}_{\rm loc}(p)$ to indicate the closure of a point p evaluated only on the local submesh.

The mesh overlap contains all points of the local mesh adjacent to points of remote meshes in the complete DAG for the parallel mesh, and we will indicate that point p is in the overlap using an indicator function $\mathcal{O}$. Moreover, if the overlap contains a point p on a given process, then it will also contain the closure of p,

$$\mathcal{O}(p) \Longrightarrow \mathcal{O}(q) \qquad \forall q \in \mathbb{cl}(p), \tag{1}$$

which shows that if a point is shared, its closure is also shared. This is a consequence of each local mesh being closed, the transitive closure of its Hasse diagram. We can now examine the effect of increasing the mesh overlap in parallel by including all the immediately adjacent mesh points to each local mesh.

The set of adjacent mesh point differs depending on the discretization. For example, the finite element method couples unknowns to all other unknowns whose associated basis functions overlap the support of the given basis function. If functions are supported on cells whose closure contains the associated mesh point, we have the relation

$$\mathbb{adj}(p, q) \Longleftrightarrow q \in \mathbb{cl}(\mathbb{st}(p)), \tag{2}$$

where we note that this relation is symmetric. For example, a degree of freedom (dof) associated with a vertex is adjacent to all dofs on the cells containing that vertex. We will call this ***FE adjacency***. On the other hand, for finite volume methods, we typically couple cell unknowns only through faces, so that we have

$$\mathbb{adj}(p, q) \Longleftrightarrow q \in \mathbb{supp}(\mathbb{cone}(p)), \tag{3}$$

which is the common notion of cell-adjacency in meshes, and what we will call ***FV adjacency***. This will also be the adjacency pattern for Discontinuous Galerkin methods.

If we first consider FV adjacency, we see that the cone operation can be satisfied locally since local meshes are closed. Thus the support from neighboring processes is needed for all points in the overlap. Moreover, in order to preserve the closure property of local meshes, the closure of that support would also need to be collected.

For FE adjacency, each process begins by collecting the star of its overlap region in the local mesh, $\mathbb{st}_{\text{loc}}(\mathcal{O})$. The union across all processes will produce the star of each point in the overlap region. First, note that if the star of a point p on the local processes contains a point q on the remote process, then q must be contained in the star of a point o in the overlap,

$$q \in \mathbb{st}(p) \iff \exists o \mid \mathcal{O}(o) \wedge q \in \mathbb{st}(o). \tag{4}$$

There is a path from p to q in the mesh DAG, since q lies in star of p, which is the transitive closure. There must be an edge in this path which connects a point on the local mesh to one on the remote mesh, otherwise the path is completely contained in the local mesh. One of the endpoints o of this edge will be contained in the overlap, since it contains all local points adjacent to remote points in the DAG. In fact, q lies in the star of o, since o lies on the path from p to q. Thus, the star of p is contained in the union of the star of the overlap,

$$\mathbb{st}(p) \in \bigcup_{o} \mathbb{st}(o). \tag{5}$$

Taking the closure of this star is a local operation, since local meshes are closed. Therefore, parallel overlap creation can be accomplished by the following sequence: each local mesh collects the closure of the star of its overlap, communicates this to its overlap neighbors, and then each neighbor augments its overlap with the new points. Moreover, no extra points are communicated, since each communicated point q is adjacent to some p on a remote process.

2.2. Data Distribution

We will recognize three basic objects describing a parallel data layout: the `Section` [Balay et al. 2014a] describing an irregular array of data and the `SF`, Star-Forest [Brown 2011], a one-sided description of shared data. A `Section` is a map from a domain of *points* to data sizes, or *ndofs*, and assuming the data is packed it can also calculate an offset for each point. This is exactly the encoding strategy used in the Compressed Sparse Row matrix format [Balay et al. 2014a]. An `SF` stores the owner for any piece of shared data which is not owned by the given process, so it is a one-sided description of sharing. This admits a very sparse storage scheme, and a scalable algorithm for assembly of the communication topology [Hoefler et al. 2010]. The third local object, a `Label`, is merely a one-to-many map between integers, that can be manipulated in a very similar fashion to a `Section` since the structure is so similar, but has better complexity for mutation operations.

A `Section` may be stored as a simple list of (*ndof*, *offset*) pairs, and the `SF` as (*ldof*, *rdof*, *rank*) triples where *ldof* is the local dof number and *rdof* is the remote dof number, which means we never need a global numbering of the unknowns. Starting with these two simple objects, we may mechanically build complex, parallel data distributions from simple algebraic combination operations. We will illustrate this process with a simple example.

Suppose we begin with a parallel cell-vertex mesh having degrees of freedom on the vertices. On each process, a `Section` holds the number of dofs on each vertex, and

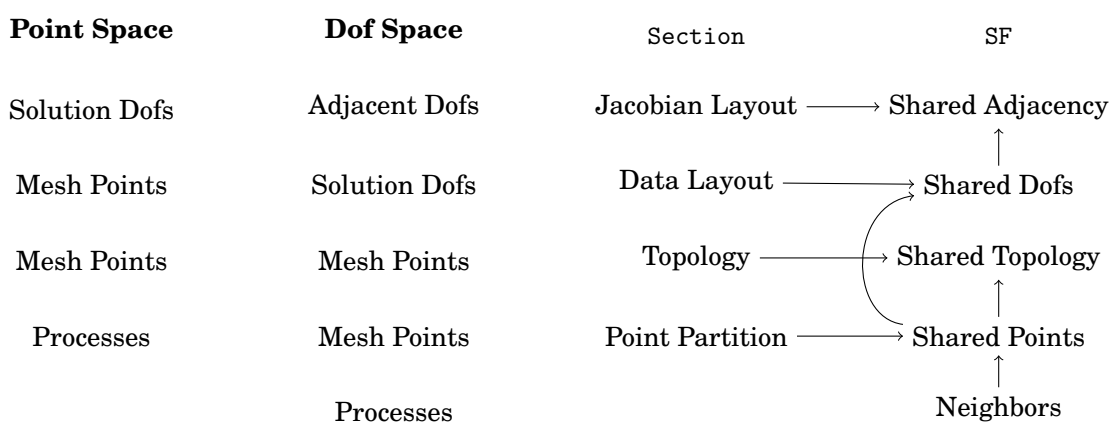

Fig. 2. This figure illustrates the relation between different `Section`/`SF` pairs. The first column gives the domain space for the `Section`, the second the range space for the `Section` and domain and range for the `SF`. The `Section` and `SF` columns give the semantic content for those structures at each level, and the arrows show how the `SF` at each level can be constructed with input from below. Each horizontal line describes the parallel layout of a certain data set. For example, the second line down describes the parallel layout of the solution field.

an `SF` lists the vertices which are owned by other processes. Notice that the domain (point space) of the `Section` is both the domain and the range (dof space) of the `SF`. We can combine these two to create a new `SF` whose domain and range (dof space) match the range space of the `Section`. This uses the `PetscSFCreateSectionSF()` function, which is completely local except for the communication of remote dof offsets, which needs a single sparse broadcast from dof owners (roots) to dof sharers (leaves), accomplished using `PetscSFBcast()`. The resulting `SF` describes the shared dofs rather than the shared vertices. We can think of the new `SF` as the push-forward along the `Section` map. This process can be repeated to generate a tower of relations, as illustrated in Fig. 2.

We can illustrate the data structures and transformations in Fig. 2 by giving concrete examples for the parallel mesh in Fig. 3. Given the partition in the figure, we have an `SF` SF_{point}, called *Shared Points* in Fig. 2,

$$SF^0_{\text{point}} = \{f \to (e,1), \epsilon \to (\beta,1), \phi \to (\gamma,1)\},$$
$$SF^1_{\text{point}} = \{e \to (f,0), \beta \to (\epsilon,0), \gamma \to (\phi,0)\},$$

where the superscript denotes the process on which the object lives. Let us define a data layout for the solution to a Stokes problem using the Taylor-Hood [Taylor and Hood 1973] finite element scheme (P_2–P_1). We define the `Section` S_u, called *Data Layout* in Fig. 2,

$$S^0_u = \{c : (2,0), d : (2,2), f : (2,4), \epsilon : (3,6), \delta : (3,9), \phi : (3,12)\}$$
$$S^1_u = \{a : (2,0), b : (2,2), e : (2,4), \alpha : (3,6), \beta : (3,9), \gamma : (3,12)\}.$$

Using `PetscSFCreateSectionSF()`, we obtain a `Section` SF_{dof}, called *Shared Dof* in Fig. 2, giving us the shared dofs between partitions,

$$SF^0_{\text{dof}} = \{4 \to (4,1), 5 \to (5,1), 6 \to (9,1), 7 \to (10,1), 8 \to (11,1),$$
$$12 \to (12,1), 13 \to (13,1), 14 \to (14,1)\}$$

which we note is only half of the relation, and SF stores one-sided information. The other half which is constructed on the fly is

$$SF^1_{\text{dof}} = \{4 \to (4,0), 5 \to (5,0), 9 \to (6,0), 10 \to (7,0), 11 \to (8,0), \\ 12 \to (12,0), 13 \to (13,0), 14 \to (14,0)\}.$$

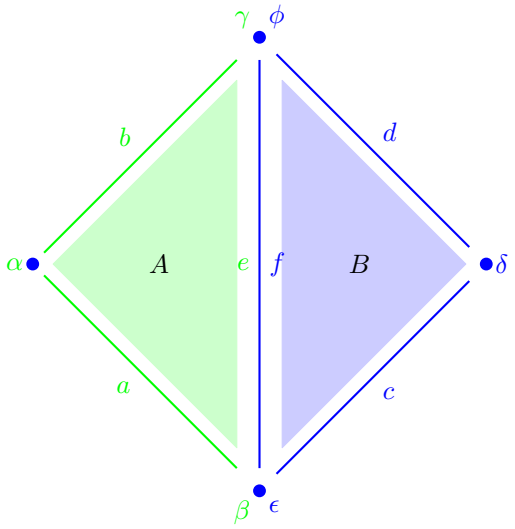

Fig. 3. A parallel simplicial doublet mesh, with points on process 0 blue and process 1 green.

We can use these same relations to transform any parallel data layout into another given an SF which connects the source and target point layouts. Suppose that we have an SF which maps currently owned points to processes which will own them after redistribution, which we will call a *migration* SF. With this SF, we can construct the section after redistribution and migrate the data itself. This process is show in Alg. 1, which uses PetscSFCreateSectionSF() from above to transform the migration SF over points to one over dofs, and also PetscSFDistributeSection() to create the section after redistribution. The section itself can be distributed using only one sparse broadcast, although we typically use another to setup remote dof offsets for PetscSFCreateSectionSF(), as shown in Alg. 2.

Algorithm 1 Algorithm for migrating data in parallel

1: **function** MIGRATEDATA(sf, secSource, dtype, dataSource, secTarget, dataTarget)
2: PETSCSFDISTRIBUTESECTION(sf, secSource, remoteOff, secTarget)
3: PETSCSFCREATESECTIONSF(sf, secSource, remoteOff, secTarget, sfDof)
4: PETSCSFBCAST(sfDof, dtype, dataSource, dataTarget)

These simple building blocks can now be used to migrate all the data for a DMPlex object, representing an unstructured mesh of arbitrary dimension composed of cells, each of which may have any shape. The migration of cone data, coordinates, and labels all follow the general migration algorithm above, since each piece of data can be expressed as the combination of a Section, giving the layout, and an array storing the values, in PETSc a Vec or IS object. Small differences from the generic algorithm arise due to

Algorithm 2 Algorithm for migrating a `Section` in parallel

```
1: function DISTRIBUTESECTION(sf, secSource, remoteOff, secTarget)
2:     <Calculate domain (chart) from local SF points>
3:     PETSCSFBCAST(sf, secSource.dof, secTarget.dof)      ▷ Move point dof sizes
4:     PETSCSFBCAST(sf, secSource.off, remoteOff)          ▷ Move point dof offsets
5:     PETSCSECTIONSETUP(secTarget)
```

the nature of the stored data. For example, the cone data must also be transformed from original local numbering to the new local numbering, which we accomplish by first moving to a global numbering and then to the new local numbering using two local-to-global renumberings. After moving the data, we can compute a new point `SF` using Alg. 4, which uses a reduction to compute the unique owners of all points.

Algorithm 3 Algorithm for migrating a mesh in parallel

```
1: function MIGRATE(dmSource, sf, dmTarget)
2:     ISLOCALTOGLOBALMAPPINGAPPLYBLOCK(l2g, csize, cones, cones)
3:                                          ▷ Convert to global numbering
4:     PETSCSFBCAST(sf, l2g, l2gMig)        ▷ Redistribute renumbering
5:     DMPLEXDISTRIBUTECONES(dmSource, sf, l2gMig, dmTarget)
6:     DMPLEXDISTRIBUTECOORDINATES(dmSource, sf, dmTarget)
7:     DMPLEXDISTRIBUTELABELS(dmSource, sf, dmTarget)
```

Algorithm 4 Algorithm for migrating an SF in parallel

```
1: function MIGRATESF(sfSource, sfMig, sfTarget)
2:     PETSCSFGETGRAPH(sfMig, Nr, Nl, leaves, NULL)
3:     for p ← 0, Nl do                ▷ Make bid to own all points we received
4:         lowners[p].rank = rank
5:         lowners[p].index = leaves ? leaves[p] : p
6:     for p ← 0, Nr do                ▷ Flag so that MAXLOC does not use root value
7:         rowners[p].rank = -1
8:         rowners[p].index = -1
9:     PETSCSFREDUCE(sfMigration, lowners, rowners, MAXLOC)
10:    PETSCSFBCAST(sfMigration, rowners, lowners)
11:    for p ← 0, Nl, Ng = 0 do
12:        if lowners[p].rank != rank then
13:            ghostPoints[Ng] = leaves ? leaves[p] : p
14:            remotePoints[Ng].rank = lowners[p].rank
15:            remotePoints[Ng].index = lowners[p].index
16:            Ng++
17:    PETSCSFSETGRAPH(sfTarget, Np, Ng, ghostPoints, remotePoints)
```

2.3. Mesh Distribution

Using the data migration routines above, we can easily accomplish sophisticated mesh manipulation in PETSc. Thus, we can redistribute a given mesh in parallel, a special case of which is distribution of a serial mesh to a set of processes. As shown in Alg. 5, we first create a partition using a third party mesh partitioner, and store it as a label,

where the target ranks are label values. We take the closure of this partition in the DAG, invert the partition to get receiver data, allowing us to create a migration `SF` and use the data migration algorithms above. The only piece of data that we need in order to begin, or bootstrap, the partition process is an `SF` which connects sending and receiving processes. Below, we create the complete graph on processes, meaning that any process could communicate with any other, in order to avoid communication to discover which processes receive from the partition. Discovery is possible and sometimes desirable, and will be incorporated in a further update.

Algorithm 5 Algorithm for distributing a mesh in parallel

```
1:  function DISTRIBUTE(dm, overlap, sf, pdm)
2:      PETSCPARTITIONERPARTITION(part, dm, lblPart)          ▷ Partition cells
3:      DMPLEXPARTITIONLABELCLOSURE(dm, lblPart)              ▷ Partition points
4:      for p ← 0, P do                                       ▷ Create process SF
5:          remoteProc[p].rank = p
6:          remoteProc[p].index = rank
7:      PETSCSFSETGRAPH(sfProc, P, P, NULL, remoteProc)
8:      DMPLEXPARTITIONLABELINVERT(dm, lblPart, sfProc, lblMig)
9:                                        ▷ Convert from senders to receivers
10:     DMPLEXPARTITIONLABELCREATESF(dm, lblMig, sfMig)
11:                                                ▷ Create migration SF
12:     DMPLEXMIGRATE(dm, sfMigration, dmParallel)            ▷ Distribute DM
13:     DMPLEXDISTRIBUTESF(dm, sfMigration, dmParallel)       ▷ Create new SF
```

We can illustrate the migration process by showing how Fig. 3 is derived from Fig. 1. We begin with the doublet mesh contained entirely on one process. In the partition phase, we first create a cell partition consisting of a `Section` $S_{\rm cpart}$ for data layout and an `IS` $\rm cpart$ holding the points in each partition,

$$S_{\rm cpart} = \{0 : (1,0), 1 : (1,1)\},$$
$$\rm cpart = \{\mathit{B}, \mathit{A}\},$$

which is converted to the equivalent `Label`, a data structure better optimized for overlap insertion,

$$L_{\rm cpart} = \{0 \to \{B\}, 1 \to \{A\}\},$$

and then we create the transitive closure. We can express this as a `Section` $S_{\rm part}$, called *Point Partition* in Fig. 2, and `IS` $\rm part$ with the partition data,

$$S_{\rm part} = \{0 : (4,0), 1 : (7,4)\},$$
$$\rm part = \{\mathit{B}, \mathit{c}, \mathit{d}, \mathit{\delta}, \mathit{A}, \mathit{a}, \mathit{b}, \mathit{e}, \mathit{\alpha}, \mathit{\beta}, \mathit{\gamma}\},$$

or as the equivalent `Label`

$$L_{\rm part} = \{0 \to \{B, c, d, \delta\}, 1 \to \{A, a, b, e, \alpha, \beta, \gamma\}\}.$$

The bootstrap `SF` $SF_{\rm proc}$, called *Neighbors* in Fig. 2, encapsulates the data flow for migration

$$SF_{\rm proc} = \{0 \to (0,0), 1 \to (1,1)\}.$$

We have a small problem in that the partition structure specifies the send information, and for an `SF` we require the receiver to specify the data to be received. Thus we need to invert the partition. This is accomplished with a single call to `DMPlexDistributeData()`

from Alg. 1, which is shown in Alg. 6. This creates a `Section` and `IS` with the receive information,

$$\begin{aligned} S^0_{\text{invpart}} &= \{0 : (4, 0)\} \\ \text{invpart} &= \{B, c, d, \delta\} \\ S^1_{\text{invpart}} &= \{0 : (7, 0)\} \\ \text{invpart} &= \{A, a, b, e, \alpha, \beta, \gamma\}. \end{aligned}$$

and then we convert them back into a `Label` L_{invpart}. This simple implementation for the complex operation of partition inversion shows the power of our flexible interface for data movement. Since the functions operate on generic representations of data (e.g. `Section`, `SF`), the same code is reused for many different mesh types and mesh/data operations, and only a small codebase needs to be maintained. In fact, the distribution (one-to-many) and redistribution (many-to-many) operations are identical except for an initial inversion of the point numbering to obtain globally unique numbers for cones.

Algorithm 6 Algorithm for inverting a partition

1: MIGRATEDATA(SF_{proc}, S_{part}, MPIU_2INT, part, S_{invpart}, invpart)

After inverting our partition, we combine L_{invpart} and SF_{proc} using `DMPlexPartitionLabelCreateSF()`, the equivalent of `PetscSFCreateSectionSF()`, to obtain the `SF` for point migration

$$\begin{aligned} SF_{\text{point}} = \{&A \to (A, 1), B \to (B, 0), \\ &a \to (a, 1), b \to (b, 1), c \to (c, 0), d \to (d, 0), e \to (e, 1), \\ &\alpha \to (\alpha, 1), \beta \to (\beta, 1), \gamma \to (\gamma, 1), \delta \to (\delta, 1)\}. \end{aligned}$$

In the final step, this `SF` is then used to migrate all the (`Section`, array) pairs in the `DMPlex`, such as cones, coordinates, and labels, using the generic `DMPlexMigrate()` function.

2.4. Overlap Generation

Following the initial distribution of the mesh, which was solely based on the partitioner output, the set of overlapping local meshes can now be derived in parallel. This derivation is performed by each process computing it's local contribution to the set of overlap points on neighboring processes, starting from an SF that contains the initial point sharing. It is important to note here that this approach performs the potentially costly adjacency search in parallel and that the search space is limited to the set of points initially shared along the partition boundary.

The algorithm for identifying the set of local point contributions to neighboring partitions is based on the respective adjacency definitions given in section 2.1. As illustrated in Alg. 7, the SF containing the initial point overlap is first used to identify connections between local points and remote processes. To add a level of adjacent points, the local points adjacent to each connecting point are added to a partition label similar to the one used during the initial migration (see Alg. 5), identifying them as now also connected to the neighboring process. Once the point donations for the first level of cell overlap are defined, further levels can be added through repeatedly finding points adjacent to the current donations.

Algorithm 7 Algorithm for computing the partition overlap

```
1: function DMPlexCreateOverlap(dm, overlap, sf, odm)
2:     DMPlexDistributeOwnership(dm, sf, rootSection, rootRank)   ▷ Derive sender information from SF
3:     for leaf ← sf.leaves do                                    ▷ Add local receive connections
4:         DMPlexGetAdjacency(sf, leaf.index, adjacency)
5:         for a ← adjacency do
6:             DMLabelSetValue(lblOl, a, leaf.rank)
7:     for p ← 0, P do                                            ▷ Add local send connections
8:         if rootSection[p] > 0 then
9:             DMPlexGetAdjacency(sf, p, adjacency)
10:            for a ← adjacency do
11:                DMLabelSetValue(lblOl, a, rootRank[p])
12:    for n ← 1, overlap do                                      ▷ Add further levels of adjacency
13:        DMPlexPartitionLabelAdjacency(lblOl, n)
```

Having established the mapping required to migrate remote overlap points, we can derive a migration `SF` similar to the one used in Alg. 5. As shown in Alg. 8, this allows us to utilize `DMPlexMigrate()` to generate the overlapping local sub-meshes, provided the migration SF also encapsulates the local point renumbering required to maintain stratification in the DMPlex DAG, meaning that cells are numbered contiguously, vertices are numbered contiguously, etc. This graph numbering shift can easily be derived from the `SF` that encapsulates the remote point contributions, thus enabling us to express local and remote components of the overlap migration in a single `SF`.

Algorithm 8 Algorithm for migrating overlap points

```
1: function DMPlexDistributeOverlap(dm, overlap, sf, odm)
2:     DMPlexCreateOverlap(dm, sf, lblOl)                  ▷ Create overlap label
3:     DMPlexPartitionLabelCreateSF(dm, lblOl, sfOl)       ▷ Derive migration SF
4:     DMPlexStratifyMigrationSF(dm, sfOl, sfMig)          ▷ Shift point numbering
5:     DMPlexMigrate(dm, sfMig, dmOl)                      ▷ Distribute overlap
6:     DMPlexDistributeSF(dm, sfMig, dmOl)                 ▷ Create new SF
```

3. RESULTS

The performance of the distribution algorithms detailed in Alg. 5 and 8 has been evaluated on the UK National Supercomputer ARCHER, a Cray XE30 with 4920 nodes connected via an Aries interconnect [1]. Each node consists of two 2.7 GHz, 12-core Intel E5-2697 v2 (Ivy Bridge) processors with 64GB of memory. The benchmarks consist of distributing a three dimensional simplicial mesh of the unit cube across increasing numbers of MPI processes (strong scaling), while measuring execution time and the total amount of data communicated per processor. The mesh is generated in memory using TetGen [Si 2015; Si 2005] and the partitioner used is Metis/ParMetis [Karypis and Kumar 1998; Karypis et al. 2005].

The performance of the partitioning and data migration components of the initial one-to-all mesh distribution, as well as the subsequent generation of the parallel overlap

[1] `http://www.archer.ac.uk/`

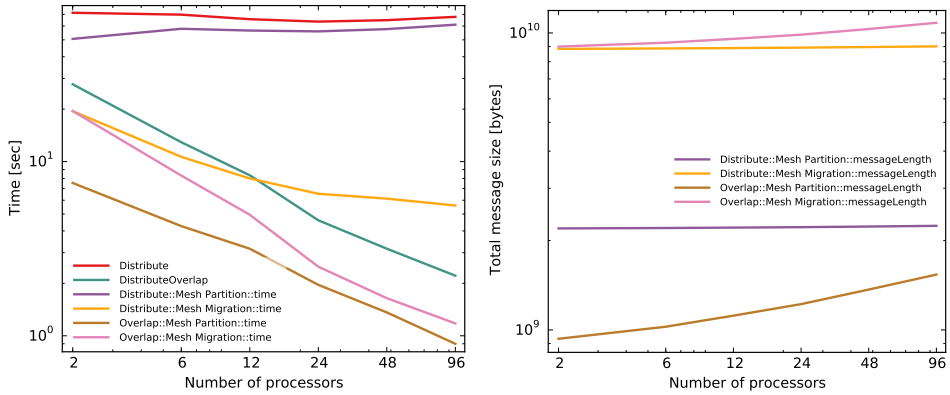

Fig. 4. Performance of initial one-to-all mesh distribution of a 3D unit cube mesh with approximately 12 million cells. The distribution time is dominated by the time to send the serial mesh to all processes, and the overlap determination and communication time scales linearly with the number of processes.

mapping is detailed in Fig. 4. The presented run-time measurements indicate that the parallel overlap generation scales linearly with increasing numbers of processes, whereas the cost of the initial mesh distribution increases due to the the sequential partitioning cost.

Data communication was measured as the accumulated message volume (sent and received) per process for each stage using PETSc's performance logging [Balay et al. 2014a]. As expected, the overall communication volume during distributed overlap generation increases with the number of processes due to data replication along the shared partition boundaries. The communicated data volume during the initial distribution, however, remains constant, indicating that the increasing run-time cost is due to sequential processing, not communication of the partitioning. In fact, the number of high-level communication calls, such as SF-broadcasts and SF-reductions is constant for meshes of all sizes and numbers of processes. A model of the total data volume communicated during the initial distribution of a three-dimensional mesh can be established as follows:

$$\begin{aligned} V_{sf} &= 4B * N \\ V_{inversion} &= V_{sf} + 2 * 4B * N \\ V_{stratify} &= V_{sf} + 4B * N \\ V_{partition} &= V_{inversion} + V_{stratify} \end{aligned} \tag{6}$$

$$
\begin{aligned}
V_{cones} &= N_c * 4B * 4 + N_f * 4B * 3 + N_e * 4B * 2 \\
V_{orientations} &= N_c * 4B * 4 + N_f * 4B * 3 + N_e * 4B * 2 \\
V_{section} &= 3 * V_{sf} + 2 * 4B * N \\
V_{topology} &= V_{cones} + V_{orientations} + V_{section} \\
V_{coordinates} &= (3 * 8B + 2 * 4B) * N_v \\
V_{markers} &= 3 * V_{sf} \\
V_{migration} &= V_{topology} + V_{coordinates} + V_{markers}
\end{aligned} \tag{7}
$$

where N_c, N_f, N_e and N_v denote the number of cells, faces, edges and vertices respectively, $N = N_c + N_f + N_e + N_v$ and V_{sf} is the data volume required to initialize an SF. The unit square mesh used in the benchmarks has $N_c = 12,582,912$, $N_f = 25,264,128$, $N_e = 14,827,904$, $N_v = 2,146,689$, resulting in $V_{partition} \approx 1.1GB$ and $V_{migration} \approx 2.8GB$.

As well as initial mesh distribution the presented API also allows all-to-all mesh distribution in order to improve load balance among partitions. Fig. 5 depicts run-time and memory measurements for such a redistribution process, where an initial bad partitioning based on random assignment is improved through re-partitioning with ParMETIS. Similarly to the overlap distribution, the run-time cost demonstrate good scalability for the partitioning as well as the migration phase, while the communication volume increases with the number of processes.

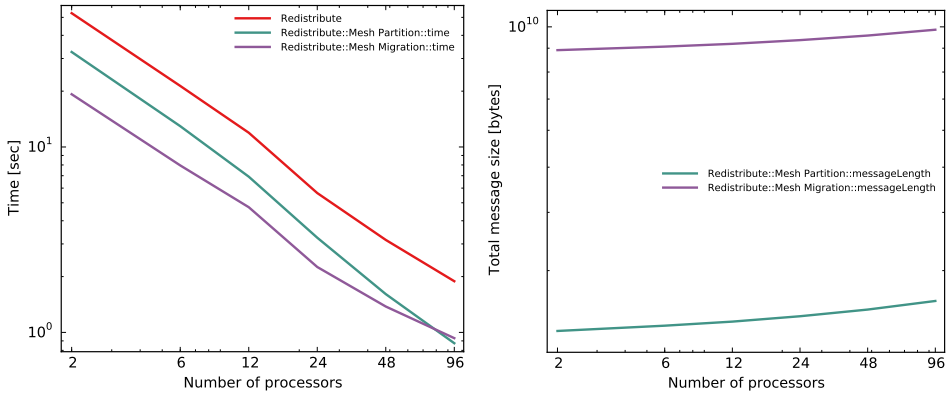

Fig. 5. Performance of all-to-all mesh distribution of simplicial meshes in 2D and 3D. An initial random partitioning is re-partitioned via ParMetis and re-distributed to achieve load balancing.

As demonstrated in Fig. 4, the sequential overhead of generating the base mesh on a single process limits overall scalability of parallel run-time mesh generation. To overcome this bottleneck, parallel mesh refinement can be used to create high-resolution meshes in parallel from an initial coarse mesh. The performance benefits of this approach are highlighted in Fig. 6, where regular refinement is applied to a unit cube mesh with varying numbers of edges in each dimension. The performance measurements show clear improvements for the sequential components, initial mesh generation and distribution, through the reduced mesh size, while the parallel refinement

operations and subsequent overlap generation scale linearly. Such an approach is particularly useful for the generation of mesh hierarchies required for multigrid preconditioning.

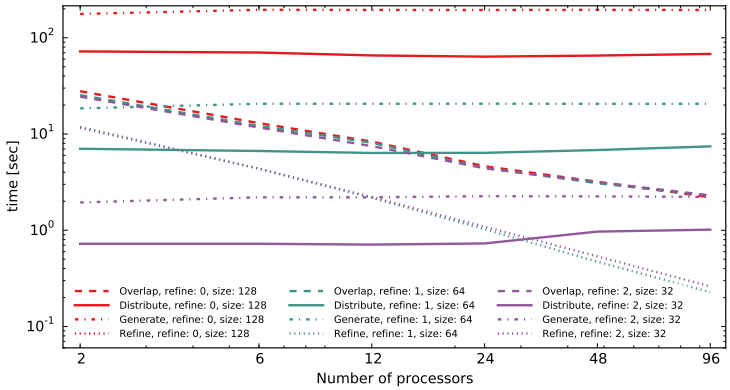

Fig. 6. Performance of parallel mesh generation via regular three dimensional refinement.

4. CONCLUSIONS

We have developed a concise, powerful API for general parallel mesh manipulation, based upon the `DMPlexMigrate()` capability. With just a few methods, we are able to express mesh distribution from serial to parallel, parallel redistribution of a mesh, and parallel determination and communication of arbitrary overlap. Moreover, a user could combine these facilities to specialize mesh distribution for certain parts of a calculation or for certain fields or solvers, since they are not constrained by a monolithic interface. Moreover, the same code applies to meshes of any dimension, with any cell shape and connectivity. Thus optimization of these few routines would apply to the universe of meshes expressible as CW-complexes. In combination with a set of widely used mesh file format readers this provides a powerful set of tools for efficient mesh management available to a wide range of applications through PETSc library interfaces [Lange et al. 2015].

In future work, we will apply these building blocks to the problem of fully parallel mesh construction and adaptivity. We will input a naive partition of the mesh calculable from common serial mesh formats, and then rebalance the mesh in parallel. We are developing an interface to the Pragmatic unstructured parallel mesh refinement package [Rokos and Gorman 2013], which will allow parallel adaptive refinement where we currently use only regular refinement.

REFERENCES

BALAY, S., ABHYANKAR, S., ADAMS, M. F., BROWN, J., BRUNE, P., BUSCHELMAN, K., EIJKHOUT, V., GROPP, W. D., KAUSHIK, D., KNEPLEY, M. G., MCINNES, L. C., RUPP, K., SMITH, B. F., AND ZHANG, H. 2014a. PETSc users manual. Tech. Rep. ANL-95/11 - Revision 3.5, Argonne National Laboratory.

BALAY, S., ABHYANKAR, S., ADAMS, M. F., BROWN, J., BRUNE, P., BUSCHELMAN, K., EIJKHOUT, V., GROPP, W. D., KAUSHIK, D., KNEPLEY, M. G., MCINNES, L. C., RUPP, K., SMITH, B. F., AND ZHANG, H. 2014b. PETSc Web page. `http://www.mcs.anl.gov/petsc`.

BIRKHOFF, G. 1967. *Lattice theory*. Vol. 25. American Mathematical Society.

BROWN, J. 2011. Star forests as a parallel communication model.

D'AZEVEDO, E. AND ET. AL. 2015. Itaps web site.

DEDNER, A., KLÖFKORN, R., NOLTE, M., AND OHLBERGER, M. 2010. A generic interface for parallel and adaptive discretization schemes: abstraction principles and the dune-fem module. *Computing 90,* 3-4, 165–196.

DEVINE, K. D., BOMAN, E. G., HEAPHY, R. T., ÇATALYÜREK, U. V., AND BISSELING, R. H. 2006. Parallel hypergraph partitioning for irregular problems. *SIAM Parallel Processing for Scientific Computing*.

HATCHER, A. 2002. *Algebraic topology*. Cambridge University Press.

HOEFLER, T., SIEBERT, C., AND LUMSDAINE, A. 2010. Scalable Communication Protocols for Dynamic Sparse Data Exchange. In *Proceedings of the 2010 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*. ACM, 159–168.

KARYPIS, G. AND KUMAR, V. 1998. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing 48*, 71–85.

KARYPIS ET AL., G. 2005. ParMETIS Web page. `http://www.cs.umn.edu/~karypis/metis/parmetis`.

KNEPLEY, M. G. AND KARPEEV, D. A. 2009. Mesh algorithms for PDE with Sieve I: Mesh distribution. *Scientific Programming 17,* 3, 215–230. `http://arxiv.org/abs/0908.4427`.

LANGE, M., KNEPLEY, M. G., AND GORMAN, G. J. 2015. Flexible, scalable mesh and data management using petsc dmplex.

ROKOS, G. AND GORMAN, G. 2013. Pragmatic–parallel anisotropic adaptive mesh toolkit. In *Facing the Multicore-Challenge III*. Springer, 143–144.

SI, H. 2005. TetGen: A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator. `http://tetgen.berlios.de`.

SI, H. 2015. Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Trans. on Mathematical Software 41,* 2.

TAYLOR, C. AND HOOD, P. 1973. A numerical solution of the navier-stokes equations using the finite element technique. *Computers & Fluids 1,* 1, 73–100.

WIKIPEDIA. 2015a. Cw complex. `http://en.wikipedia.org/wiki/CW_complex`.

WIKIPEDIA. 2015b. Hasse diagram. `http://en.wikipedia.org/wiki/Hasse_diagram`.